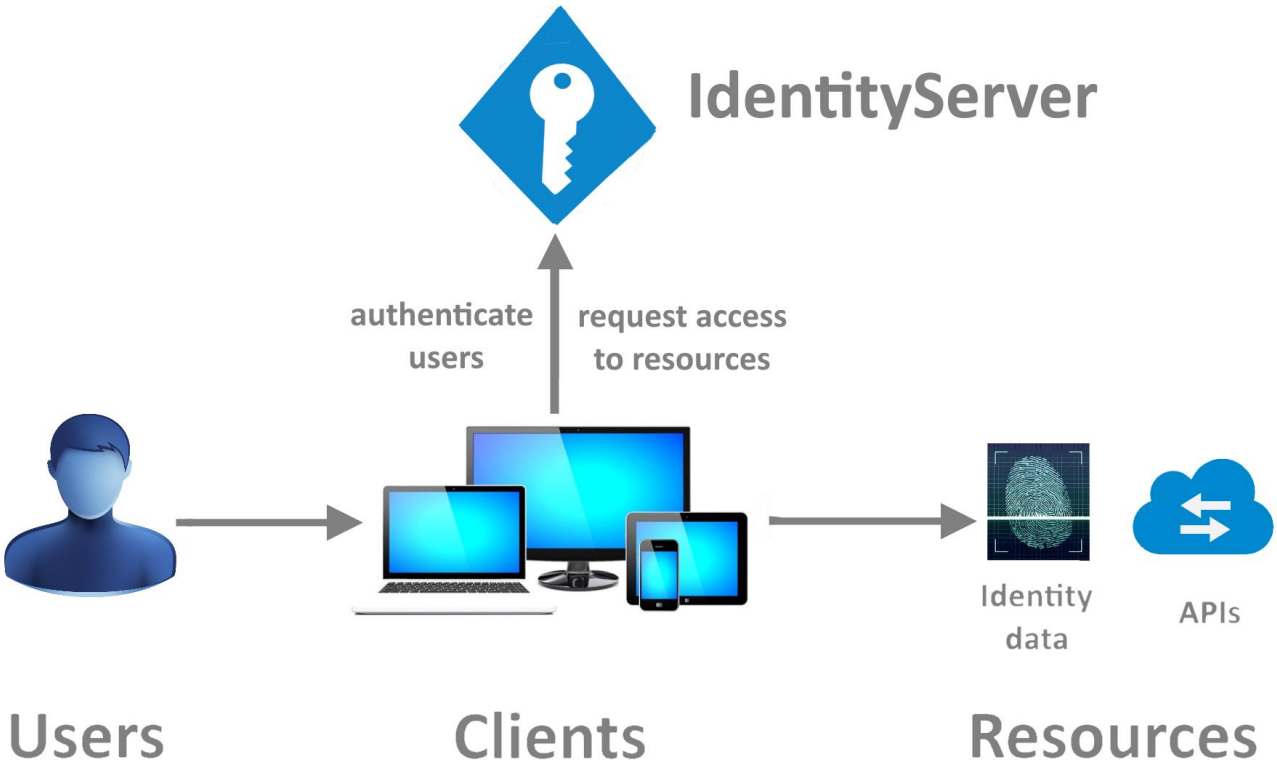
Руководство:
Защита и доступ к API
с использованием
Identity Server for BTP4ru

Содержание:

ЗНАКОМСТВО С IDENTITY SERVER

The big Picture или Кратко: как тут все устроено?



- *IdentityServer* (identity provider) определяет параметры защиты ресурсов, регистрирует допустимые клиенты и выдает токены безопасности клиентам;
- *Пользователь* использует зарегистрированный клиент для запроса у IdentityServer идентификации и разрешений на доступ к защищенной информации и ресурсам;
- *Клиент* запрашивает у IdentityServer токены безопасности. Предварительно должен быть зарегистрирован на IdentityServer;
- *Ресурсы* доступны только тем клиентам и/или пользователям, которые имеют соответствующий типу ресурса токен и необходимое на доступ разрешение (scope). Токен должен быть получен у источника, которому ресурс доверяет (IdentityServer).
- *Токен* является совокупностью утверждений (claims) о пользователе и клиенте:
  - *identity_token* – содержит информацию для идентификации пользователя;
  - *access_token* – содержит разрешения (scopes) для определения доступа к ресурсам;
- *Утверждение* является составной частью токена и используется ресурсом для определения прав доступа (principal) пользователя и/или клиента*.

* – подробнее процесс *авторизации на основе утверждений* рассмотрим **далее**

Введение в Claims-based identity

Авторизация, основанная на утверждениях – это подход, при котором решение авторизации о предоставлении или запрете доступа определенному пользователю базируется на произвольной логике, которая в качестве входных данных использует некий набор утверждений *(`claims`)*, относящихся к этому пользователю. Проводя аналогию с Role-based подходом, у пользователя в его наборе `claims` будет только один элемент с типом `role`, который используется для проверки принадлежит ли пользователю какая-либо определенная роль (например, роль `administrator`).

Разберем по шагам процесс принятия решения механизмом авторизации в ASP.NET Core с использованием claims-based подхода:

1) Приложение получает запрос, который требует идентификации пользователя;
2) ASP.NET Core перенаправляет пользователя к identity provider (в нашем случае – на Identity Server). После успешной идентификации пользователя, в клиентский запрос добавляются токены безопасности: описывающий пользователя (*identity_token*), и описывающий права пользователя на доступ к ресурсам (*access_token*), в виде утверждений (*claims*) о нем. Затем ASP.NET Core связывает claims с правами доступа (*principal*) пользователя;
3) Приложение передает claims в механизм логики принятия решения. Это может быть встроенный код, вызов web-сервиса, запрос к базе данных, механизм составных правил или использование ClaimsAuthorizationManager;
4) Механизм принятия решения вычисляет результат, основанный на утверждениях (claims);
5) Доступ предоставляется, если результат вычисления `true`, и отвергается, если `false`. (Например, правило гласит, что возраст пользователя должен быть 21 или более лет и его место жительства – Москва).

Простым языком claims-based identity можно описать на следующем примере:

Вам исполнилось 18 и вы решили сходить в кино. Взрослое кино. Но, не успели получить паспорт (или любое другое удостоверение личности) до сих пор. Вот вы идете в паспортный стол, через какое-то время получаете паспорт и, предъявляя ваш паспорт, смело покупаете себе заветный билетик и идете на сеанс. А вот так это выглядит изнутри:

`User`, то есть вы как посетитель (`Client`) сеанса для взрослых, идете к `Identity Provider` (паспортный стол) и на основе `Credentials` (`СвидетельствоОРождении`Token) получаете `ПаспортToken`. Затем вместе с `ПаспортToken` вы идете в `Resource` (кинотеатр) и, после подтверждения вашего возраста (`"age" claim`), получаете доступ к услуге.

Основные идеи, которые можно извлечь из этого примера:

1) Для авторизации вас, как посетителя сеанса для взрослых, кинотеатру не требуется вести свою базу клиентов или обращаться куда-либо. Ему достаточно вашего удостоверения личности, которому он доверяет;
2) Паспортный стол не знает где именно (из списка зарегистрированных поставщиков товаров и услуг) вы будете предъявлять паспорт;
3) С паспортом вы можете купить себе хорошего виски после похода в кино, взять ипотеку, слетать за границу или еще что-то в любом учреждении, которое доверяет документам, выданным гос. учреждением.

Дополнительная информация:

- An Introduction to Claims;
- Claims-based identity term definitions;
- Security in .NET 4.5: Claims & Tokens become the standard Model

БЕЗОПАСНОСТЬ API

API

API (Application Programming Interface) — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) или операционной системой для использования во внешних программных продуктах. Используется программистами при написании всевозможных приложений.

API определяет функциональность, которую предоставляет программа (модуль, библиотека), при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована.

Если программу (модуль, библиотеку) рассматривать как чёрный ящик, то API — это множество «ручек», которые доступны пользователю данного ящика и которые он может вертеть и дёргать.

Программные компоненты взаимодействуют друг с другом посредством API. При этом обычно компоненты образуют иерархию — высокоуровневые компоненты используют API низкоуровневых компонентов, а те, в свою очередь, используют API ещё более низкоуровневых компонентов.

Аутентификация и авторизация

- ✓ **Аутентификация, идентификация** (authentication) используется для надежного определения личности пользователя;
- ✓ **Авторизация** (authorization) используется для определения ресурсов, к которым идентифицированный пользователь имеет доступ;

Аутентификация и авторизация обычно используются вместе.

В сети аутентификация чаще всего осуществляется через диалоговое окно с запросом имени пользователя и пароля. Для API же обычно используется токен доступа (*access_token*), полученный через внешний процесс (например, при регистрации для API) или через отдельный механизм (например, OAuth2). Этот тип токенов используется в системе, когда приложение должно получать доступ к API от имени пользователя. Токен доступа инкапсулирует разрешения доступа пользователя для приложения при запросах API. Токен передается с каждым запросом к API и проверяется API перед обработкой запроса.

Json Web Token

JSON Web Token (JWT) – это открытый стандарт (RFC 7519), который определяет компактный и автономный способ безопасной передачи информации между сторонами в виде объекта *JSON*. Эта информация может быть проверена и доверена, поскольку она имеет цифровую подпись.

Аутентификация – это наиболее распространенный сценарий использования JWT. После входа в систему каждый последующий запрос будет включать JWT, что позволит пользователю получать доступ к маршрутам, службам и ресурсам, которые разрешены этим токеном. Single Sign On – функция, которая широко использует JWT в настоящее время из-за ее небольших накладных расходов и возможности легко использовать в разных доменах.

При аутентификации, когда пользователь успешно выполнит вход в систему, используя свои учетные данные, будет возвращен JWT, который нужно сохранить локально (в LocalStorage) вместо традиционного подхода с созданием сеанса на сервере и возвращением файла cookie.

Всякий раз, когда пользователь захочет получить доступ к защищенному маршруту или ресурсу, клиент должен отправить JWT, как правило, в заголовке `Authorization` используя схему `Bearer`. Содержимое заголовка должно выглядеть следующим образом:

```
Authorization: Bearer <token>
```

Это механизм аутентификации без состояния, поскольку пользовательское состояние никогда не сохраняется в памяти сервера. Защищенные маршруты сервера будут проверять наличие действительного JWT в заголовке авторизации, и если он присутствует, пользователю будет разрешен доступ к защищенным ресурсам. JWT являются автономным и вся необходимая информация содержится внутри, что уменьшает необходимость многократных запросов к базе данных.

Это позволяет вам полностью полагаться на API, не имеющие состояния, и даже отправлять запросы на службы нижестоящего уровня. Не имеет значения, какие домены обслуживают ваши API, так что Cross-Origin Resource Sharing (*CORS*) не будет проблемой, так как JWT не используют cookie.

JSON Web Token состоит из трех частей, разделенных точкой:

- Header (заголовок);
- Payload (основная часть, тело);
- Signature (подпись);

Следовательно, JWT выглядит так:

```
xxxxx.yyyyy.zzzzz
```

Дополнительная информация:

- Подробнее о JWT: https://jwt.io/introduction/
- Валидатор JWT: https://jwt.io/#debugger

Шаг 1. Конфигурация API

1.1. Создать **проект** Visual Studio 2015 по типу ASP.NET Core Web Application.

1.2. В **project.json** добавить пакет:

```
"IdentityServer4.AccessTokenValidation"
```

1.3. В **Startup.cs** в метод `Configure()` перед `app.UseMvc()` добавить middleware:

```
var authOptions = new IdentityServerAuthenticationOptions
{
    Authority = "https://localhost:5000",  // Identity Server address
    ApiName = "api1",
    ApiSecret = "__API-SECRET__",
    AllowedScopes = { "api1.full_access", "api1.read_only" },
    SupportedTokens = SupportedTokens.Jwt,  // JWT, Reference or Both
    SaveToken = true,
    AutomaticAuthenticate = true,
    AutomaticChallenge = true,
    RequireHttpsMetadata = false,
}
app.UseIdentityServerAuthentication(authOptions);
```

1.4. К **контроллеру**, который требуется защитить, добавить атрибут `Authorize`:

```
using Microsoft.AspNetCore.Authorization;

namespace SecuredApi.Controllers
{
    [Authorize]
    [Route("secured")]
    public class SecuredController : Controller
    {
        // ...
    }
}
```

1.5. `(optional)` Если клиентское приложение на JavaScript

В **Startup.cs** в метод `Configuration()` добавить конфигурацию *CORS*:

```
services.AddCors(options =>
{
    options.AddPolicy("default", policy =>
    {
    policy.WithOrigins("http://localhost:5003") // адрес приложения на JavaScript
            .AllowAnyHeader()
            .AllowAnyMethod();
    });
});
```

В метод `Configure()` добавить:

```
app.UseCors("default");
```

Шаг 2. Конфигурация Identity Server

2.1. В Configurations **API Resource Scopes** создать новые scope для всех указанных в конфигурации API на шаге 1.3 => AllowedScopes (далее они помещаются в API-ресурс на шаге 2.2):

```
// Имя, указанное в конфигурации API
```
Name = "api1.full_access"

```
// Отображаемое имя
```
DisplayName = "API_1 Admin"

```
// Подробное описание
```
Description = "API_1 administrator's full access scope"

```
// Список claim'ов, которые будут добавлены в access_token
// Ввести каждый на отдельной строке
```
UserClaims = { }

2.2. В Configurations **API Resources** создать новый ресурс API, задав ему список доступных scope'ов:

```
// Разрешение на доступ клиенту к scope'ам, включенным в данный API-ресурс
```
Enabled = true

```
// Имя API, указанное в конфигурации
```
Name = "api1 "

```
// Отображаемое имя
```
DisplayName = "My API 1"

```
// Подробное описание
```
Description = "My Protected API Resource"

```
// Список секретов для защиты API
// Ввести каждый на отдельной строке
```
ApiSecrets = { "__API-SECRET__" }

```
// Список claim'ов, которые будут добавлены в access_token
// Ввести каждый на отдельной строке
```
UserClaims = { "name", "role", "email" }

```
// Список scope'ов, которые будут доступны в данном API
// Отметить нужные галочкой
```
Scopes = { "api1.full_access", "api1.read_only" }

2.3. В Configurations **Clients** нужному клиенту добавить доступные ему scopes:

```
// Список scope'ов API, которые будут доступны данному клиенту
// Отметить нужные галочкой
```
AllowedScopes = { "openid", "profile", "offline_access", " api1.full_access", " api1.read_only" }

Шаг 3. Конфигурация Client

В IdentityServer определены различные виды клиента. Детали процесса создания и настройки специфичны и описаны подробно в разделе **Клиенты** для каждого вида отдельно.

В общем случае, конфигурация клиента и взаимодействие с API выглядят следующим образом:

3.1. Определить клиент в конфигурации Identity Server;

3.2. Создать новый проект и подключить вспомогательные библиотеки;

3.3. `(optional)` *для некоторых видов клиента этот шаг не требуется.*
Добавить авторизацию и идентификацию и настроить последнюю, используя параметры, указанные в конфигурации данного клиента на Identity Server;

3.4. Авторизовать клиент и/или пользователя на Identity Server и получить access_token;

3.5. Добавить access_token в http-запросах к API в заголовок для авторизации запросов в API:
`Authorization: Bearer <access_token>`

3.6. Отправить запрос и получить ответ от API.

Далее рассмотрим эти шаги для каждого вида клиента.

КЛИЕНТЫ

Вжух! И краткое описание появилось здесь.

Server-to-server

Для создания клиента типа "server-to-server" используется Client Credentials grant type. Это самый простой grant type, в котором токены запрашиваются от имени клиента, а не пользователя.

*Клиент запрашивает доступ у TokenEndpoint, используя **ClientId** и **ClientSecret**, и в ответ получает **access_token**.*

1. Определение клиента:

1.1. **Авторизуйтесь** в IdentityServer, используя Ваши логин и пароль;

1.2. Перейдите в Configurations \ **Clients**;

1.3. Нажмите на кнопку **Создать**;

1.4. Задайте следующие **параметры**:

```
// ID клиента
ClientId = "comp2comp"

// Имя клиента
ClientName = "BTP4ru comp to comp client"

// Тип клиента
// Выберите из списка
AllowedGrantTypes = { "client_credentials" }

// Секрет (шифруется в sha256)
// Отметьте галочку “set or change?”
ClientSecrets = { "__CLIENT-SECRET__" }

// Список scope’ов API, которые будут доступны данному клиенту
// Отметить нужные галочкой
AllowedScopes = { "openid", "profile", "api1.full_access" }
```

1.5. Сохраните изменения, нажав на кнопку **Сохранить**;

2. Создание клиента:

2.1. Создать **проект** Visual Studio 2015 по типу ASP.NET Core Web Application.

2.2. В **project.json** добавить пакет:

```
"IdentityModel"
```

2.3. Создать **контроллер** для взаимодействия с API, добавить **атрибут** `Authorize` и следующие **зависимости**:

```
using System;
using System.Net.Http;
using System.Net.Http.Formatting;
using System.Threading.Tasks;
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using IdentityModel.Client;

namespace ClientCompToComp.Controllers
{
    [Authorize]
    public class ApiInteractionController : Controller
    {
        // ...
    }
}
```

3. Идентификация и авторизация

Для данного вида клиента не требуется.

4. Получение токена

4.1. Добавить в контроллер метод:

```
private async Task<TokenResponse> RequestTokenAsync()
{
    // конфигурация клиента для получения discovery-документа Identity Server
    var discovery = await DiscoveryClient.GetAsync("http://localhost:5000");
    if (discovery.IsError) throw new Exception(discovery.Error);

    // конфигурация клиента для получения access_token
    var client = new TokenClient(discovery.TokenEndpoint, "comp2comp", "__SECRET__");

    // ответ содержит access_token
    return await client.RequestClientCredentialsAsync("api1.full_access");
}
```

5. Защита http-запросов

5.1. Добавить в контроллер метод:

```
private HttpClient ConfigureClient(string apiUrl, string token)
{
    // конфигурация клиента
    var client = new HttpClient { BaseAddress = new Uri(apiUrl) };
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json")
    );

    // добавляем access_token в заголовок запроса
    client.SetBearerToken(token);

    return client;
}
```

6. Взаимодействие с API

6.1. Добавить в контроллер методы:

```
private Task<ContentType> GetFormattedContentAsync(HttpResponseMessage response)
{
    if (!response.IsSuccessStatusCode)
        throw new Exception($"API call fails with status: {response.StatusCode}");

    // задаем форматирование для содержимого ответа
    var formatters = new List<MediaTypeFormatter>()
    {
        new JsonMediaTypeFormatter()
    };

    // получаем содержимое из ответа API с форматированием
    return response.Content.ReadAsAsync<ContentType>(formatters);
}

public async Task<IActionResult> CallApiUsingAccessToken()
{
    // запрашиваем access_token
    var tokenResponse = await RequestTokenAsync();
    if (tokenResponse.IsError) throw new Exception(
        $"Token Endpoint call fails with status: {tokenResponse.HttpStatusCode}",
        tokenResponse.Exception);

    // получаем access_token
    var accessToken = response.AccessToken;

    // создаем клиент
    var client = ConfigureClient("http://localhost:5001", accessToken);

    // отправляем запрос к защищенному контроллеру API
    var response = await client.GetAsync("secured");

    // получаем содержимое ответа
    var content = await GetFormattedContentAsync(response);

    return new JsonResult(content);
    //return View(content);
}
```

JavaScript

Для создания клиента на JavaScript используется Implicit grant type. Этот тип оптимизирован под браузерные (JavaScript-) приложения для аутентификации и запроса access_token.

*Клиент аутентифицируется, используя **UserLogin** и **UserPassword**, и в ответ получает **id_token** и **access_token**. Токены передаются через браузер; **refresh_token** (Offline Access) и **ClientSecret** – не поддерживаются.*

1. Определение клиента:

1.1. Авторизуйтесь в IdentityServer, используя Ваши логин и пароль;

1.2. Перейдите в Configurations \ **Clients**;

1.3. Нажмите на кнопку **Создать**;

1.4. Задайте следующие **параметры**:

```
// ID клиента
ClientId = "javascript"

// Имя клиента
ClientName = "BTP4ru JavaScript client"

// Тип клиента
// Выберите из списка
AllowedGrantTypes = { "implicit" }

// Разрешить передачу access_token через браузер
AllowAccessTokensViaBrowser = true

// Разрешить показ пользователю Consent Screen для выбора получаемых scope'ов
RequireConsent = false

// Список scope'ов API, которые будут доступны данному клиенту
// Отметить нужные галочкой
AllowedScopes = { "openid", "profile", "api1.full_access" }

// Список допустимых ссылок для перенаправления после авторизации
// Ввести каждый на отдельной строке
RedirectUris = { "http://localhost:5003/callback.html" }

// Список допустимых ссылок для перенаправления после выхода из системы
// Ввести каждый на отдельной строке
PostLogoutRedirectUris = { "http://localhost:5003/index.html" }

// Список допустимых CORS для возможности Ajax-запросов с клиента к API
// Ввести каждый на отдельной строке
AllowedCorsOrigins = { "http://localhost:5003" }
```

2. Создание клиента:

2.1. Создать **проект** Visual Studio 2015 по типу ASP.NET Core Web Application.

2.2. В **project.json** добавить пакет:

```
"Microsoft.AspNetCore.StaticFiles"
```

2.3. В **Startup.cs** в метод `Configure()` добавить:

```
public void Configure(IApplicationBuilder app)
{
    // after logging...

    app.UseDefaultFiles();
    app.UseStaticFiles();

    // ...before authentication and mvc
}
```

2.4. В **package.json** в секцию `devDependencies` добавить пакет:

```
"oidc-client"
```

2.5. Перенести файл **`oidc-client.js`**
- из папки: `~/node_modules/oidc-client/dist`
- в папку: `~/wwwroot`

2.6. Добавить файл **index.html** в папку ~/wwwroot

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>JS Client</title>
</head>
<body>
    <button id="login">Login</button>
    <button id="api">Call API</button>
    <button id="logout">Logout</button>
    <pre id="results"></pre>
    <script src="oidc-client.js"></script>
    <script src="app.js"></script>
</body>
</html>
```

2.7. Добавить файл **callback.html** в папку ~/wwwroot

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <script src="oidc-client.js"></script>
    <script>
        new Oidc.UserManager().signinRedirectCallback()
            .then(function () {
                window.location = "index.html";
            })
            .catch(function (e) {
                console.error(e);
            });
    </script>
</body>
</html>
```

2.8. Добавить файл **app.js** в папку ~/wwwroot

```
function log() {
    document.getElementById('results').innerText = '';

    Array.prototype.forEach.call(arguments, function (msg) {
        if (msg instanceof Error) {
            msg = "Error: " + msg.message;
        }
        else if (typeof msg !== 'string') {
            msg = JSON.stringify(msg, null, 2);
        }
        document.getElementById('results').innerHTML += msg + '\r\n';
    });
}

document.getElementById("login").addEventListener("click", login, false);
document.getElementById("api").addEventListener("click", api, false);
document.getElementById("logout").addEventListener("click", logout, false);
```

3. Идентификация и авторизация:

3.1. В файл `app.js` добавить конфигурацию и функции:

```
var config = {
    authority: "http://localhost:5000", // адрес IdentityServer
    client_id: "javascript",
    redirect_uri: "http://localhost:5003/callback.html",
    response_type: "id_token token",
    scope: "openid profile api1.full_access",
    post_logout_redirect_uri: "http://localhost:5003/index.html",
};
var mgr = new Oidc.UserManager(config);

function login() {
    mgr.signinRedirect();
}

function logout() {
    mgr.signoutRedirect();
}
```

4. Получение токена:

4.1. В файл `app.js` добавить функцию:

```
mgr.getUser().then(function (user) {
    if (user) {
        log("User logged in", user.profile);
    }
    else {
        log("User not logged in");
    }
});
```

5. Защита http-запросов:

5.1. В файл `app.js` добавить функцию:

```
function getSecuredXhr() {
    var xhr = new XMLHttpRequest();
    xhr.setRequestHeader("Authorization", "Bearer " + user.access_token);
    return xhr;
}
```

6. Взаимодействие с API:

6.1. В конфигурации приложения API добавить адрес JavaScript-приложения в список разрешенных *CORS*, как показано в **конфигурации API, шаг 1.5**.

6.2. В файл `app.js` добавить функцию:

```
function api() {
    mgr.getUser().then(function (user) {
        var url = "http://localhost:5001/secured";
        var xhr = getSecuredXhr();
        xhr.open("GET", url);
        xhr.onload = function () {
            log(xhr.status, JSON.parse(xhr.responseText));
        }
        xhr.send();
    });
}
```

JavaScript and MVC

Для создания клиента на JavaScript используется Implicit grant type. Этот тип оптимизирован под браузерные (JavaScript-) приложения для аутентификации и запроса access_token.

*Клиент аутентифицируется, используя **UserLogin** и **UserPassword**, и в ответ получает **id_token** и **access_token**. Токены передаются через браузер; **refresh_token** (Offline Access) и **ClientSecret** – не поддерживаются.*

1. Определение клиента:

1.1. Авторизуйтесь в IdentityServer, используя Ваши логин и пароль;

1.2. Перейдите в Configurations \ **Clients**;

1.3. Нажмите на кнопку **Создать**;

1.4. Задайте следующие **параметры**:

```
// ID клиента
ClientId = "jsmvc"

// Имя клиента
ClientName = "BTP4ru JavaScript with MVC client"

// Тип клиента
// Выберите из списка
AllowedGrantTypes = { "implicit" }

// Разрешить передачу access_token через браузер
AllowAccessTokensViaBrowser = true

// Разрешить показ пользователю Consent Screen для выбора получаемых scope'ов
RequireConsent = false

// Список scope'ов API, которые будут доступны данному клиенту
// Отметить нужные галочкой
AllowedScopes = { "openid", "profile", "api1.full_access" }

// Список допустимых ссылок для перенаправления после авторизации
// Ввести каждый на отдельной строке
RedirectUris = { "http://localhost:5004/callback.html" }

// Список допустимых ссылок для перенаправления после выхода из системы
// Ввести каждый на отдельной строке
PostLogoutRedirectUris = { "http://localhost:5004/index.html" }

// Список допустимых CORS для возможности Ajax-запросов с клиента к API
// Ввести каждый на отдельной строке
AllowedCorsOrigins = { "http://localhost:5004" }
```

2. Создание клиента:

2.1. Создать **проект** Visual Studio 2015 по типу ASP.NET Core Web Application.

2.2. В **project.json** добавить пакеты:

```
"Microsoft.AspNetCore.StaticFiles"
"Microsoft.AspNetCore.Authentication.JwtBearer"
"Microsoft.AspNetCore.Authentication.OpenIdConnect"
"System.IdentityModel.Tokens.Jwt"
"IdentityServer4.AccessTokenValidation"
"IdentityModel"
```

2.3. Создать **контроллер** для взаимодействия с API, добавить **атрибут** `Authorize` и следующие **зависимости**:

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Net.Http.Formatting;
using System.Threading.Tasks;
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authentication;

namespace ClientJsMvc.Controllers
{
    [Authorize]
    public class ApiInteractionController : Controller
    {
        // ...
    }
}
```

2.4. В **Startup.cs** в метод `Configure()` добавить:

```
public void Configure(IApplicationBuilder app)
{
    // after logging...

    app.UseDefaultFiles();
    app.UseStaticFiles();

    // ...before authentication and mvc
}
```

2.5. В **package.json** в секцию `devDependencies` добавить пакет:

```
"oidc-client"
```

2.6. Перенести файл **`oidc-client.js`**
- из папки: `~/node_modules/oidc-client/dist`
- в папку: `~/wwwroot`

2.7. Добавить файл **index.html** в папку ~/wwwroot

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>JS Client</title>
</head>
<body>
    <button id="login">Login</button>
    <button id="api">Call API</button>
    <button id="logout">Logout</button>
    <pre id="results"></pre>
    <script src="oidc-client.js"></script>
    <script src="app.js"></script>
</body>
</html>
```

2.8. Добавить файл **callback.html** в папку ~/wwwroot

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <script src="oidc-client.js"></script>
    <script>
        new Oidc.UserManager().signinRedirectCallback()
            .then(function () {
                window.location = "index.html";
            })
            .catch(function (e) {
                console.error(e);
            });
    </script>
</body>
</html>
```

2.9. Добавить файл **app.js** в папку ~/wwwroot

```
function log() {
    document.getElementById('results').innerText = '';

    Array.prototype.forEach.call(arguments, function (msg) {
        if (msg instanceof Error) {
            msg = "Error: " + msg.message;
        }
        else if (typeof msg !== 'string') {
            msg = JSON.stringify(msg, null, 2);
        }
        document.getElementById('results').innerHTML += msg + '\r\n';
    });
}

document.getElementById("login").addEventListener("click", login, false);
document.getElementById("api").addEventListener("click", api, false);
document.getElementById("logout").addEventListener("click", logout, false);

var config = {
    authority: "http://localhost:5000", // адрес IdentityServer
    client_id: "jsmvc",
    redirect_uri: "http://localhost:5004/callback.html",
    response_type: "id_token token",
    scope: "openid profile api1.full_access",
    post_logout_redirect_uri: "http://localhost:5004/index.html",
};
var mgr = new Oidc.UserManager(config);

mgr.getUser().then(function (user) {
    if (user) {
        log("User logged in", user.profile);
    }
    else {
        log("User not logged in");
    }
});

function login() {
    mgr.signinRedirect();
}

function api() {
    mgr.getUser().then(function (user) {
        var url = "http://localhost:5004/api/secured"; // адрес метода на сервере aspnet mvc

        var xhr = new XMLHttpRequest();
        xhr.open("GET", url);
        xhr.onload = function () {
            log(xhr.status, JSON.parse(xhr.responseText));
        }
        // добавляем access_token в заголовок get-запроса
        xhr.setRequestHeader("Authorization", "Bearer " + user.access_token);
        xhr.send();
    });
}

function logout() {
    mgr.signoutRedirect();
}
```

3. Идентификация и авторизация:

3.1. В **Startup.cs** добавить зависимости:

```
using System.IdentityModel.Tokens.Jwt;
using Newtonsoft.Json.Serialization;
```

3.2. В метод `ConfigureServices()` добавить:

```
public void ConfigureServices(IServiceCollection services)
{
    // ...other configurations here...

    // добавляем авторизацию пользователей
    services.AddAuthorization();

    // добавляем сервисы MVC и сериализатор json для автоформатирования ответа клиенту
    services.AddMvc()
        .AddJsonOptions(options =>
        {
            options.SerializerSettings.ContractResolver =
                new CamelCasePropertyNamesContractResolver();
        });
}
```

3.3. В метод `Configure()` добавить:

```
public void Configure(
    IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory
    )
{
    // ...other configurations here...

    // очищаем claim type mappings
    JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

    // идентификация пользователя с использованием токена
    var authOptions = new IdentityServerAuthenticationOptions
    {
        Authority = "http://localhost:5000", // адрес IdentityServer
        AllowedScopes = { "api1.full_access" },
        SaveToken = true,
        RequireHttpsMetadata = false,
        AutomaticAuthenticate = true,
        AutomaticChallenge = true,
    };
    app.UseIdentityServerAuthentication(authOptions);

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");

        routes.MapSpaFallbackRoute(
            name: "spa-fallback",
            defaults: new { controller = "Home", action = "Index" });
    });
}
```

4. Получение токена

4.1. Добавить в контроллер метод:

```
private Task<string> RequestTokenAsync()
{
    return HttpContext.Authentication.GetTokenAsync("access_token");
}
```

5. Защита http-запросов

5.1. Добавить в контроллер метод:

```
private HttpClient ConfigureClient(string apiUrl, string token)
{
    // конфигурация клиента
    var client = new HttpClient { BaseAddress = new Uri(apiUrl) };
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json")
    );

    // добавляем access_token в заголовок запроса
    client.SetBearerToken(token);

    return client;
}
```

6. Взаимодействие с API

6.1. Добавить в контроллер методы:

```
private Task<ContentType> GetFormattedContentAsync(HttpResponseMessage response)
{
    if (!response.IsSuccessStatusCode)
        throw new Exception($"API call fails with status: {response.StatusCode}");

    // задаем форматирование для содержимого ответа
    var formatters = new List<MediaTypeFormatter>()
    {
        new JsonMediaTypeFormatter()
    };

    // получаем содержимое из ответа API с форматированием
    return response.Content.ReadAsAsync<ContentType>(formatters);
}


[HttpGet("api/secured")]
public async Task<IActionResult> CallApiUsingAccessToken()
{
    // получаем access_token
    var accessToken = await RequestTokenAsync();

    // создаем клиент
    var client = ConfigureClient("http://localhost:5001", accessToken);

    // отправляем запрос к защищенному контроллеру API
    var response = await client.GetAsync("secured");

    // получаем содержимое ответа
    var content = await GetFormattedContentAsync(response);

    return new JsonResult(content);
    //return View(content);
}
```

ASP.NET Core Web Application

Для создания клиента на ASP.NET Core Web Application (MVC) используется Hybrid flow. Этот тип является комбинацией типов Implicit и AuthorizationCode flow и использует различные типы ответа, обычно: `code id_token`. В Hybrid flow id_token содержит подписанный ответ протокола, что предотвращает множество атак, производимых через канал браузера.

*Клиент аутентифицируется, используя **UserLogin** и **UserPassword**, и после успешной валидации ответа, получает **id_token** через канал браузера, а также **access_token** и **refresh_token** через обратный канал.*

1. Определение клиента:

1.1. Авторизуйтесь в IdentityServer, используя Ваши логин и пароль;

1.2. Перейдите в Configurations **Clients**;

1.3. Нажмите на кнопку **Создать**;

1.4. Задайте следующие **параметры**:

```
// ID клиента
ClientId = "mvc"

// Имя клиента
ClientName = "BTP4ru MVC client"

// Тип клиента
// Выберите из списка
AllowedGrantTypes = { "hybrid" }

// Секрет (шифруется в sha256)
// Отметьте галочку “set or change?”
ClientSecrets = { "__CLIENT-SECRET__" }

// Разрешить показ пользователю Consent Screen для выбора получаемых scope’ов
RequireConsent = false

// Разрешить получение refresh_token (для обновления access_token)
AllowOfflineAccess = true

// Список scope’ов API, которые будут доступны данному клиенту
// Отметить нужные галочкой
AllowedScopes = { "openid", "profile", "api1.full_access", "offline_access" }

// Список допустимых ссылок для перенаправления после авторизации
// Ввести каждый на отдельной строке
RedirectUris = { "http://localhost:5005/signin-oidc" }

// Список допустимых ссылок для перенаправления после выхода из системы
// Ввести каждый на отдельной строке
PostLogoutRedirectUris = { "http://localhost:5005" }
```

1.5. Сохраните изменения, нажав на кнопку **Сохранить**;

2. Создание клиента:

2.1. Создать **проект** по типу ASP.NET Core Web Application.

2.2. В **project.json** добавить пакеты:

```
"Microsoft.AspNetCore.Authentication.Cookies"
"Microsoft.AspNetCore.Authentication.OpenIdConnect"
"IdentityModel"
```

2.3. Создать **контроллер** для взаимодействия с API, добавить **атрибут** `Authorize` и следующие **зависимости**:

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Net.Http.Formatting;
using System.Threading.Tasks;
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using IdentityModel.Client;

namespace ClientMvc.Controllers
{
    [Authorize]
    public class ApiInteractionController : Controller
    {
        // ...
    }
}
```

3. Идентификация и авторизация

3.1. В **Startup.cs** добавить зависимости:

```
using System.IdentityModel.Tokens.Jwt;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;
```

3.2. В метод `ConfigureServices()` добавить:

```
public void ConfigureServices(IServiceCollection services)
{
    // ...other configurations here...

    // добавляем авторизацию пользователей
    services.AddAuthorization();

    // добавляем сервисы MVC
    services.AddMvc();
}
```

3.3. В метод `Configure()` добавить:

```
public void Configure(
    IApplicationBuilder app, IHostingEnvironment env
    )
{
    // ...other configurations here...

    // очищаем claim type mappings
    JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

    // идентификация пользователя с использованием cookie
    app.UseCookieAuthentication(new CookieAuthenticationOptions
    {
        AuthenticationScheme = CookieAuthenticationDefaults.AuthenticationScheme
    });

    // hybrid flow
    var authOptions = new OpenIdConnectOptions
    {
        AuthenticationScheme = OpenIdConnectDefaults.AuthenticationScheme,
        SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme,

        Authority = "http://localhost:5000", // адрес IdentityServer
        RequireHttpsMetadata = false,

        ClientId = "mvc",
        ClientSecret = "__CLIENT-SECRET__",

        ResponseType = OpenIdConnectResponseType.CodeIdToken, // code id_token
        Scope = { OpenIdConnectScope.OpenIdProfile, "api1.full_access", "offline_access" },

        GetClaimsFromUserInfoEndpoint = true,
        SaveTokens = true,
    };

    app.UseOpenIdConnectAuthentication(authOptions);

    app.UseMvcWithDefaultRoute();
}
```

3.4. В **контроллер** добавить зависимости и метод `Logout()`:

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;


// controller

public async Task<IActionResult> Logout()
{
    // logout клиента локальный (browser cookies)
    await HttpContext.Authentication.SignOutAsync(
        CookieAuthenticationDefaults.AuthenticationScheme);

    // logout клиента глобальный (на стороне IdentityServer)
    await HttpContext.Authentication.SignOutAsync(
        OpenIdConnectDefaults.AuthenticationScheme);

    return Redirect("~/");
}
```

4. Получение токена

4.1. Добавить в контроллер метод:

```
private Task<string> RequestTokenAsync()
{
    return HttpContext.Authentication.GetTokenAsync("access_token");
}
```

5. Защита http-запросов

5.1. Добавить в контроллер метод:

```
private HttpClient ConfigureClient(string apiUrl, string token)
{
    // конфигурация клиента
    var client = new HttpClient { BaseAddress = new Uri(apiUrl) };
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json")
    );

    // добавляем access_token в заголовок запроса
    client.SetBearerToken(token);

    return client;
}
```

6. Взаимодействие с API

6.1. Добавить в контроллер методы:

```
private Task<ContentType> GetFormattedContentAsync(HttpResponseMessage response)
{
    if (!response.IsSuccessStatusCode)
        throw new Exception($"API call fails with status: {response.StatusCode}");

    // задаем форматирование для содержимого ответа
    var formatters = new List<MediaTypeFormatter>()
    {
        new JsonMediaTypeFormatter()
    };

    // получаем содержимое из ответа API с форматированием
    return response.Content.ReadAsAsync<ContentType>(formatters);
}

public async Task<IActionResult> CallApiUsingAccessToken()
{
    // получаем access_token
    var accessToken = await RequestTokenAsync();

    // создаем клиент
    var client = ConfigureClient("http://localhost:5001", accessToken);

    // отправляем запрос к защищенному контроллеру API
    var response = await client.GetAsync("secured");

    // получаем содержимое ответа
    var content = await GetFormattedContentAsync(response);

    return new JsonResult(content);
    //return View(content);
}
```

ГЛОССАРИЙ

IdentityServer

IdentityServer is an OpenID Connect provider - it implements the OpenID Connect and OAuth 2.0 protocol.

Different literature uses different terms for the same role - you probably also find security token service, identity provider, authorization server, IP-STS and more.

But they are in a nutshell all the same: a piece of software that issues security tokens to clients.

IdentityServer has a number of jobs and features - including:

- ✓ protect your resources
- ✓ authenticate users using a local account store or via an external identity provider
- ✓ provide session management and single sign-on
- ✓ manage and authenticate clients
- ✓ issue identity and access tokens to clients
- ✓ validate tokens

User

A user is a human that is using a registered client to access resources.

Client

A client is a piece of software that requests tokens from IdentityServer - either for authenticating a user (requesting an identity token) or for accessing a resource (requesting an access token). A client must be first registered with IdentityServer before it can request tokens.

Examples for clients are web applications, native mobile or desktop applications, SPAs, server processes etc.

Resources

Resources are something you want to protect with IdentityServer - either identity data of your users, or APIs.

Every resource has a unique name - and clients use this name to specify to which resources they want to get access to.

Identity data Identity information (aka claims) about a user, e.g. name or email address.

APIs APIs resources represent functionality a client wants to invoke - typically modelled as Web APIs, but not necessarily.

Identity Token

An identity token represents the outcome of an authentication process. It contains at a bare minimum an identifier for the user (called the `sub` aka subject claim) and information about how and when the user authenticated. It can contain additional identity data.

Access Token

An access token allows access to an API resource. Clients request access tokens and forward them to the API. Access tokens contain information about the client and the user (if present). APIs use that information to authorize access to their data.

Refresh Token

Refresh token allow gaining long lived access to APIs.

You typically want to keep the lifetime of access tokens as short as possible, but at the same time don't want to bother the user over and over again with doing a front-channel roundtrips to IdentityServer for requesting new ones.

Refresh tokens allow requesting new access tokens without user interaction. Every time the client refreshes a token it needs to make an (authenticated) back-channel call to IdentityServer. This allows checking if the refresh token is still valid, or has been revoked in the meantime.

Refresh tokens are supported in hybrid, authorization code and resource owner password flows. To request a refresh token, the client needs to include the `offline_access` scope in the token request (and must be authorized to for that scope).

JWT

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

JSON

JavaScript Object Notation is an open-standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is the most common data format used for asynchronous browser/server communication, largely replacing XML, and is used by AJAX.

JSON is a language-independent data format. It was derived from JavaScript, but as of 2017 many programming languages include code to generate and parse JSON-format data. The official Internet media type for JSON is `application/json`. JSON filenames use the extension `.json`.

JSON is built on two structures:

- ✓ A collection of *name/value pairs*. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- ✓ An *ordered list* of values. In most languages, this is realized as an array, vector, list, or sequence.

Claim

A claim is a statement that one subject makes about itself or another subject. The statement can be about a name, identity, key, group, privilege, or capability, for example. Claims are issued by a provider, and they are given one or more values and then packaged in security tokens that are issued by an issuer, commonly known as a *security token service* (STS) or *identity provider*. They are also defined by a claim value type and, possibly, associated metadata.

CORS

Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources (e.g. fonts) on a web page to be requested from another domain outside the domain from which the first resource was served. A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos. Certain "cross-domain" requests, notably AJAX requests, however are forbidden by default by the same-origin security policy.

CORS defines a way in which a browser and server can interact to determine whether or not it is safe to allow the cross-origin request. It allows for more freedom and functionality than purely same-origin requests, but is more secure than simply allowing all cross-origin requests. It is a recommended standard of the W3C.